

# Emulator 8086

LINK

<http://www.emu8086.com>

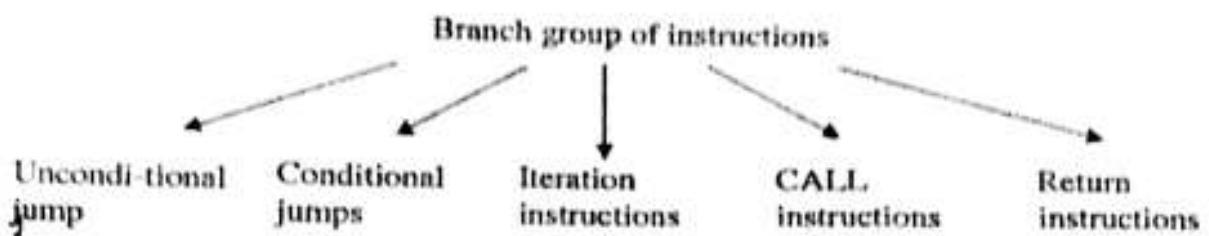
Download free evaluation from here

Key

Name: Cracked-By-Team-AggressioN!!!

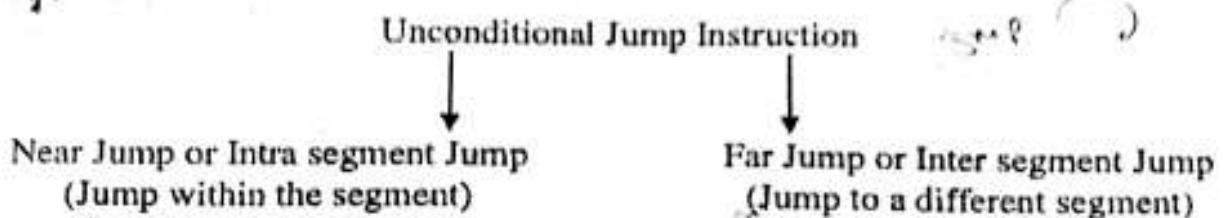
Serial: KDLSR2ERKRG4U8PDP4UA

## Flow control instruction



## Jump Group

### 1- Unconditional Jumps (JMP)



#### a- direct jumps

- intrasegment jumps
  - ① - short jump
  - ② - near jump
- intersegment jump
  - ③ - far jump

#### b- indirect jumps

- jumps with register operands
- jumps using an index

### 2- Conditional Jumps

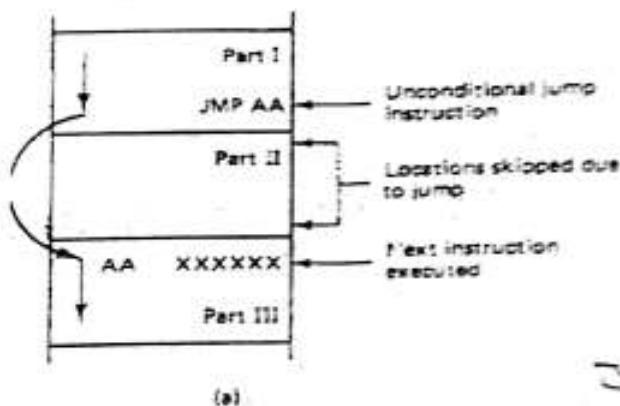
### 3- Loops and Conditional Loops

## 1- Unconditional Jump

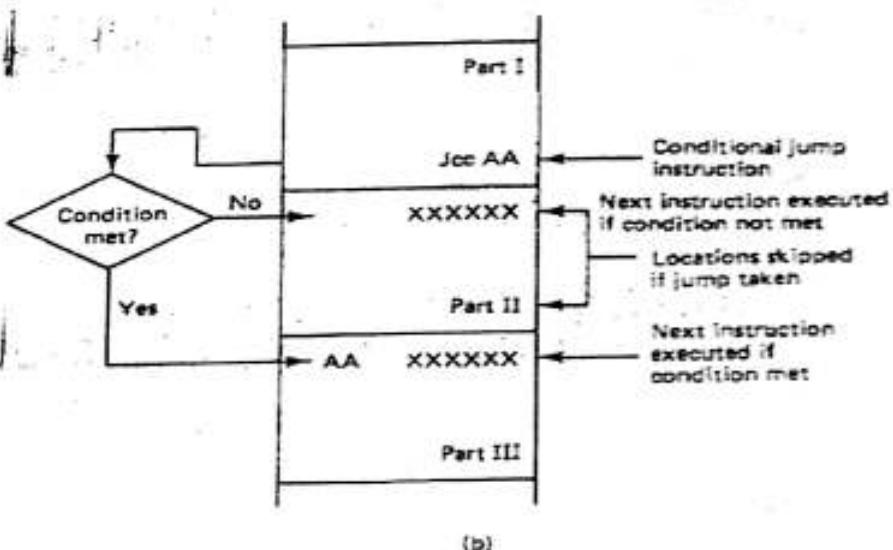
### Jump Instructions

Format: **JMP Disp**

- There are two types of jump, **unconditional** and **conditional**.  
Use both conditional and unconditional jump instructions to control the flow of a program



JMP - control  
2. unconditional



(a) Unconditional jump program sequence

(b) Conditional jump program sequence.

## Unconditional Jump

Mnemonic	Meaning	Format	Operation	Flags affected
JMP	Unconditional Jump	JMP Operand	Jump is initiated to the address specified by the operand	None

Operands
Short-label
Near-label
Far-label
Memptr16
Regptr16
Memptr32

Allowed operands for JMP instruction

### JMP Target

- Unconditional jump
- It moves microprocessor to execute another part of the program.
- Target can be represented by a label, immediate data, registers, or memory locations.
- It does not affect flags

#### Examples:

JMP BX ; IP will take the value in BX

JMP [BX] ; IP will take the value in memory location pointed to by BX

JMP DWORD PTR [DI] ; DS:DI points to two words in memory, the first word identifies the new IP and the next word identifies the new CS.

JMP 06h ; (IP)  $\leftarrow$  (IP) + 06h.

JMP 0340h ; (IP)  $\leftarrow$  (IP) + 0340h.

JMP 2040:3050 ; (IP)  $\leftarrow$  3050h, (CS)  $\leftarrow$  2040h.

JMP [BX] ; (IP)  $\leftarrow$  [DS: (BX)]

JMP CX ; (IP)  $\leftarrow$  (CX)

JMP NEAR PTR[DI] ; (IP)  $\leftarrow$  [DS: (DI)]

JMP FAR PTR[BP] ; (IP)  $\leftarrow$  [SS: (BP)], (CS)  $\leftarrow$  [SS: (BP)+2]

- Three types: short jump, near jump, far jump.

**Short jump** is a 2-byte instruction that allows jumps or branches to memory locations within  $\pm 127$  and  $-128$  bytes. From the address following the jump.

**3-byte near jump** allows a branch or jump within  $\pm 32K$  bytes from the instruction in the current code segment.

**5-byte far jump** allows a jump to any memory location within the real memory system.

The short and near jumps are often called **intrasegment jumps**. Far jumps are called **intersegment jumps**.

#### The three main forms of the JMP instruction.

Note that Disp is either an 8- or 16-bit signed displacement or distance.

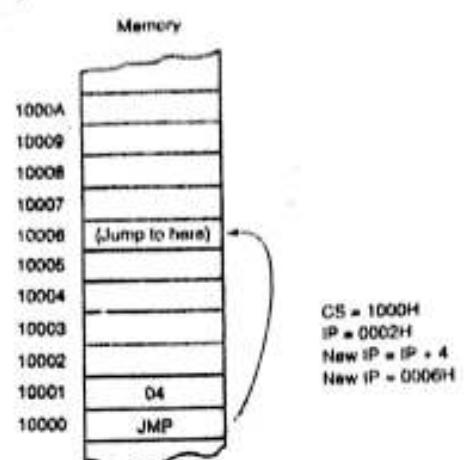
Opcode	Short		Far	
(a) EB	Disp			

Opcode	Near		
(b) E 9	Disp Low	Disp High	

Opcode	IP Low	IP High	CS Low	CS High	Far
(c) EA					

## Short Jump

- Called relative jumps because they can be moved, with related software, to any location in the current code segment without a change.
- jump address is not stored with the opcode. A distance, or displacement, follows the opcode.
- The short jump displacement is a distance represented by a 1-byte signed number whose value ranges between +127 and -128.
- Short jump instruction appears in figure below.
- When the microprocessor executes a short jump, the displacement is sign-extended and added to the instruction pointer (IP) to generate the jump address within the current code segment.
- The instruction branches to this new address for the next instruction in the program. When a jump references an address, a label normally identifies the address.



- The JMP NEXT instruction is an example. it jumps to label NEXT for the next instruction
- very rare to use an actual hexadecimal address with any jump instruction.
- The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it. if a colon does not follow, you cannot jump to it
- The only time a colon is used is when the label is used with a jump or call instruction.

## SHORT Assembler Directive

Assembler generates only 2 byte Short jump code for forward jump, if the SHORT assembler directive is used.

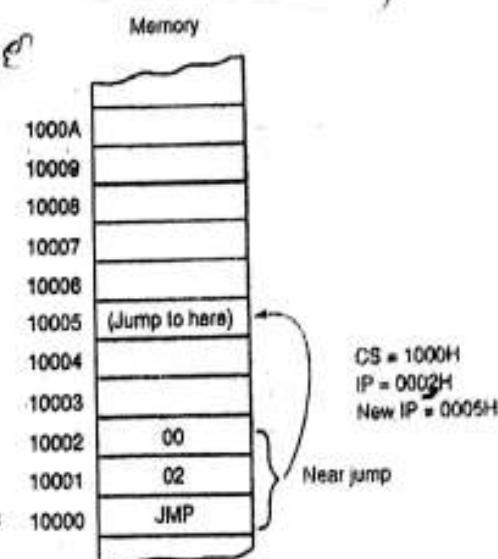
JMP SHORT SAME

Programmer should ensure that  
the Jump distance is <=127 bytes

{ SAME: MOV CX, DX

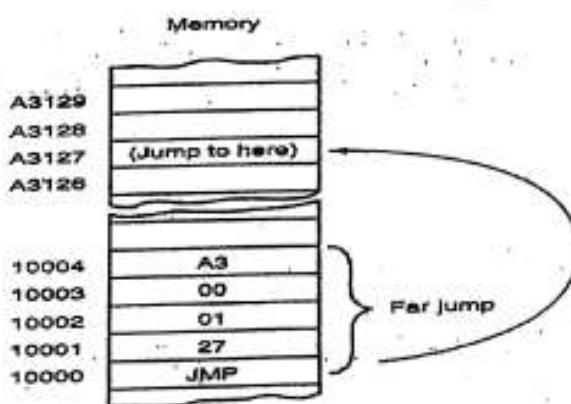
### Near Jump

- A near jump passes control to an instruction in the current code segment located within  $\pm 32K$  bytes from the near jump instruction.
- Near jump is a 3-byte instruction with opcode followed by a signed 16-bit displacement.
- Signed displacement adds to the instruction pointer (IP) to generate the jump address. because signed displacement is  $\pm 32K$ , a near jump can jump to any memory location within the current real mode code segment
- The figure below illustrates the operation of the real mode near jump instruction. A near jump that adds the displacement (0002H) to the contents of IP.



### Far Jump

- Obtains a new segment and offset address to accomplish the jump:
- bytes 2 and 3 of this 5-byte instruction contain the new offset address
- bytes 4 and 5 contain the new segment address



- The far jump instruction sometimes appears with the FAR PTR directive.
- another way to obtain a far jump is to define a label as a far label .a label is far only if it is external to the current code segment or procedure.

### Jumps with Register Operands

- Jump can also use a 16 bit register as an operand.
- automatically sets up as an indirect jump
- address of the jump is in the register specified by the jump instruction
- Unlike displacement associated with the near jump, register contents are transferred directly into the instruction pointer.
- An indirect jump does not add to the instruction pointer.
- allows a jump to any location within the current code segment

- **JMP BX**, for example

#### **JMP BX**

- This instruction replaces the content of IP with the content of BX.
- BX must first be loaded with the offset of the destination instruction in CS.
- This is a near jump. It is also referred to as an **indirect jump** because the new value of IP comes from a register rather than from the instruction itself, as in a direct jump.

### **Indirect Jumps Using an Index**

- Jump instruction may also use the [ ] form of addressing to directly access the jump table.
- The jump table can contain offset addresses for near indirect jumps, or segment and offset addresses for far indirect jumps.

#### **JMP WORD PTR [BX]**

This instruction replaces IP with word from a memory location pointed to by BX in DX. This is an **indirect near jump**.

#### **JMP DWORD PTR [SI]**

This instruction replaces IP with word pointed to by SI in DS. It replaces CS with a word pointed by SI +2 in DS. This is an **indirect far jump**.

JMP  
far  $\rightarrow$  214  
near  $\rightarrow$  -123 -127

JMP A

JMP D

CS:DX

Ex.1: JMP DX

If DX = 1234H, branches to CS:1234H. 1234H is not signed relative displacement.

Ex. 2: JMP wordptr 2000H[BX]

If BX contents is 1234H

Branches to CS:5678H

JMP to 1234H + 2000H

= 3234H

DS:3234H

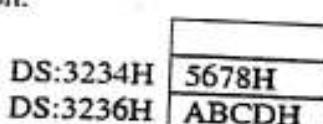
5678H

DS:3236H

AB22H

Ex. JMP DWORD PTR 2000H[BX]

If BX contents is 1234H branch takes place to location ABCDH:5678H. It is a 4-byte instruction.

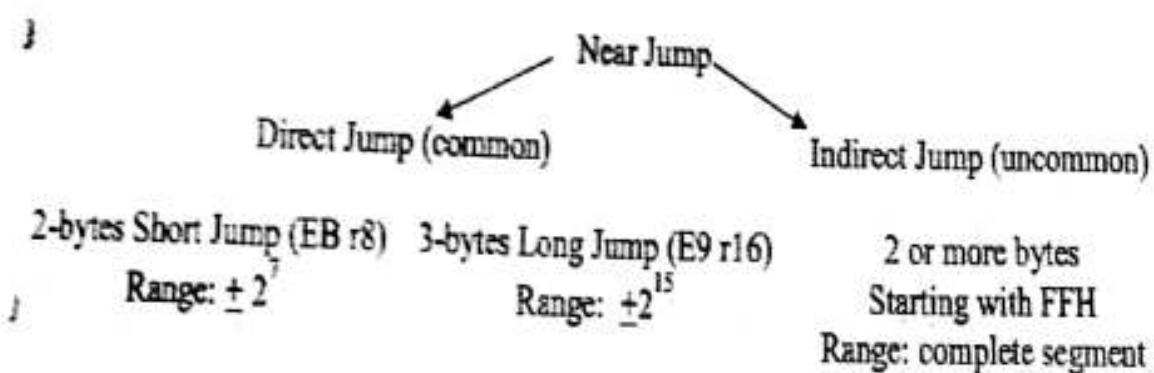


### Direct Jump v.s. Indirect Jump

- Direct Jump: the target address is directly given in the instruction
- Indirect Jump: the target address is contained in a register or memory location

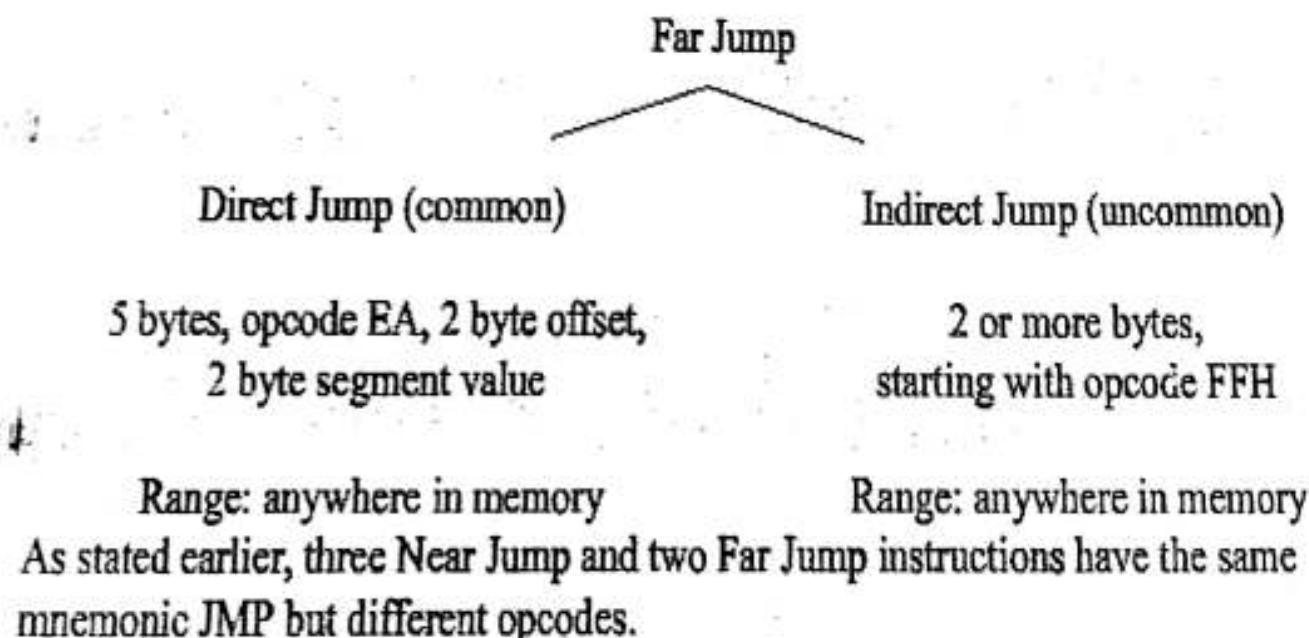
NEAR procedure	FAR procedure
A near call is a call to procedure which is in same code segment	A far call is a call to procedure which is in different segment
The contents of CS is not stored	The contents of CS is also stored along with offset.
In near call contents of SP is decremented by 2 and contents of offset address IP is stored	In Far call contents of SP are decremented by 2 and value of CS is loaded. Then SP is again decremented by 2 and IP is loaded.

### Near Unconditional Jump instruction



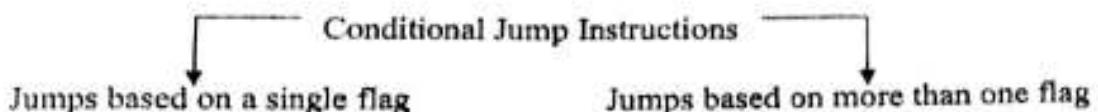
Three Near Jump and two Far Jump instructions have the same mnemonic JMP, but they have different opcodes

## Far Jump instruction



## 2- Conditional Jump

Conditional Jump instructions in 8086 are just 2 bytes long. 1-byte opcode followed by 1-byte signed displacement (range of -128 to +127).



Mnemonic	Meaning	Format	Operation	Flags affected
Jcc	Conditional Jump	Jcc Operand	If the specified condition cc is true the jump to the address specified by the operand is initiated; otherwise the next instruction is executed	None

- Conditional Jump instructions are always short jump. limits range to within +127 and -128 bytes from the location following the conditional jump.
- Allows a conditional jump to any location within the current code segment.
- Conditional Jump Instructions Jumps based on a single flag or more than one flag. **Sign (S) - Zero (Z) - Carry (C) - Overflow (O) - Parity (P)**
- If the condition under test is true, a branch to the label associated with the jump instruction occurs. if false, next sequential step in program executes
- When *signed numbers* are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions. terms greater than and less than refer to signed numbers
- When *unsigned numbers* are compared, use the JA, JB, JAE, JBE, JE, and JNE instructions. terms above and below refer to unsigned numbers
- Remaining conditional jumps test individual flag bits, such as overflow and parity. notice that JE has an alternative opcode JZ.

### Jumps Based on a single flag

JZ r8	;Jump if zero flag set (if result is 0). JE also means same.
JNZ r8	;Jump if Not Zero. JNE also means same.
JS r8	;Jump if Sign flag set to 1 (if result is negative)
JNS r8	;Jump if Not Sign (if result is positive)
JC r8	;Jump if Carry flag set to 1. JB and JNAE also mean same.
JNC r8	;Jump if No Carry. JAE and JNB also mean same.
JP r8	;Jump if Parity flag set to 1. JPE (Jump if Parity Even) also means same.
JNP r8	;Jump if No Parity. JPO (Jump if Parity Odd) also means same.
JO r8	;Jump if Overflow flag set to 1 (if result is wrong)
JNO r8	;Jump if No Overflow (if result is correct)

JZ, JNZ, JC and JNC used after arithmetic operation

JE, JNE, JB, JNAE, JAE and JNB are used after a compare operation.

### Terms used in comparison

Above and Below used for comparing Unsigned numbers. Greater than and less than used when comparing signed numbers. All Intel microprocessors use this convention.

Accordingly, all the following statements are true.

95H is above 65H	Unsigned comparison - True
95H is less than 65H	Signed comparison - True (as 95H is negative, 65H is positive)
65H is below 95H	Unsigned comparison - True
65H is greater than 95H	Signed comparison - True

### Jump based on multiple flags

Conditional Jumps based on multiple flags are used after a CMP (compare) instruction.

### JBE / JNA instruction

'Jump if Below or Equal' or 'Jump if Not Above'

Jump if	No Jump if	Ex.
Cy = 1 OR Z = 1 Below OR Equal	Cy = 0 AND Z = 0 Surely Above	CMP BX, CX JBE BX_BE

BX\_BE (BX is Below or Equal) is a symbolic location

## Examples for JE or JZ instruction

### Ex. for forward jump

Only examples using JE instruction given for forward and backward jumps.

Should be  $\leq 127$  bytes

CMP SI, DI	
JE SAME	
ADD CX, DX	; Executed if Z = 0 (if SI not equal to DI)
:	
;	
SAME: SUB BX, AX	; Executed if Z = 1 (if SI = DI)

### Ex. for backward jump

Should be  
 $\leq 127$  bytes

BACK: SUB BX, AX	; Executed if Z = 1 (if SI=DI)
:	
:	
CMP SI, DI	
JE BACK	
ADD CX, DX	; Executed if Z = 0 (if SI $\neq$ DI)

6.3  
Type of conditional jump instructions

Mnemonic	Meaning	Condition
JA	Above	CF=0 and ZF=0
JAE	Above or equal	CF=0
JB	Below	CF=1
JBE	Below or equal	CF=1 or ZF=1
JC	Carry	CF=1
JCXZ	CX register is zero	(CF or ZF)=0
JE	Equal	ZF=1
JG	Greater	ZF=0 and SF=OF
JGE	Greater or equal	SF=OF
JL	Less	(SF xor OF)=1
JLE	Less or equal	((SF xor OF) or ZF)=1
JNA	Not above	CF=1 or ZF=1
JNAE	Not above nor equal	CF=1
JNB	Not below	CF=0
JNBE	Not below nor equal	CF=0 and ZF=0

Mnemonic	Meaning	Condition
JNC	Not carry	CF=0
JNE	Not equal	ZF=0
JNG	Not greater	((SF xor OF) or ZF)=1
JNGE	Not greater nor equal	(SF xor OF)=1
JNL	Not less	SF=OF
JNLE	Not less or nor equal	ZF=0 and SF=OF
JNO	Not overflow	OF=0
JNP	Not parity	PF=0
JNS	Not sign	SF=0
JNZ	Not zero	ZF=0
JO	Overflow	OF=1
JP	Parity	PF=1
JPE	Parity even	PF=1
JPO	Parity odd	PF=0
JS	Sign	SF=1
JZ	Zero	ZF=1

**Opcode****JMP****Unconditional****Jump****JE or JZ****ZF = 1****Jump if Equal or Jump if Zero (=)****JNE or JNZ****ZF = 0****Jump if Not Equal or Jump if Not Zero****Signed Comparisons****JG****ZF = 0 and SF = 0** Jump if Greater than (>)**JGE****SF = 0****Jump if Greater than or Equal (>=)****JL****SF = 1****Jump if Less than (<)****JLE****ZF = 1 or SF = 1****Jump if Less than or Equal (<=)****Unsigned Comparisons****JA****ZF = 0 and CF = 0** Jump if Above (>)**JAE****CF = 0****Jump if Above or Equal (>=)****JB****CF = 1****Jump if Below (<)****JBE****ZF = 1 or CF = 1****Jump if Below or Equal (<=)****Miscellaneous****JO****OF = 1****Jump if Overflow, ditto for CF, SF & PF****JNO****OF = 0****Jump if No Overflow, ditto for ...****JCXZ****CX = 0****Jump if CX = 0****JECXZ****ECX = 0****Jump if ECX = 0**

<b>JZ, JE</b>	Jump If Zero (Equal).	ZF = 1	JNZ, JNE
<b>JC, JB, JNAE</b>	Jump If Carry (Below, Not Above Equal).	CF = 1	JNC, JNS, JAE
<b>JS</b>	Jump If Sign.	SF = 1	JNS
<b>JO</b>	Jump If Overflow.	OF = 1	JNO
<b>JPE, JPO</b>	Jump If Parity Even.	PF = 1	JPO
<b>JNZ, JNE</b>	Jump If Not Zero (Not Equal).	ZF = 0	JZ, JE
<b>JNC, JNS, JAE</b>	Jump If Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
<b>JNS</b>	Jump If Not Sign.	SF = 0	JS
<b>JNO</b>	Jump If Not Overflow.	OF = 0	JO
<b>JPO, JNP</b>	Jump If Parity Odd (No Parity).	PF = 0	JPE, JP

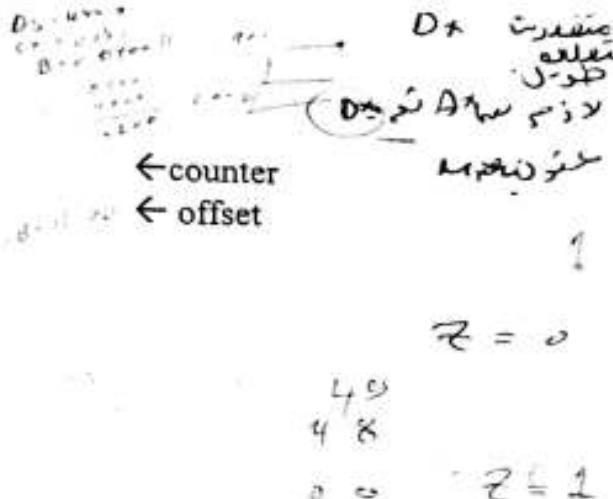
	Description	Condition	
<b>JE, JZ</b>	Jump If Equal (=). Jump If Zero.	ZF = 1	JNE, JNZ
<b>JNE, JNZ</b>	Jump If Not Equal (<>). Jump If Not Zero.	ZF = 0	JE, JZ
<b>JG, JNLE</b>	Jump if Greater (>). Jump if Not Less or Equal (not $\leq$ ).	ZF = 0 et SF = OF	JNG, JLE
<b>JL, JNGE</b>	Jump If Less (<). Jump If Not Greater or Equal (not $\geq$ ).	SF $\neq$ OF	JNL, JGE
<b>JGE, JNL</b>	Jump If Greater or Equal ( $\geq$ ). Jump If Not Less (not <).	SF = OF	JNGE, JL
<b>JLE, JNG</b>	Jump if Less or Equal ( $\leq$ ). Jump if Not Greater (not >).	ZF = 1 ou SF $\neq$ OF	JNLE, JG

**Example :** Write a program to add (50)H numbers stored at memory locations start at 4400:0100H , then store the result at address 200H in the same data segment.

```

        MOV AX, 4400H
        MOV DS, AX
        MOV CX, 0050H
        MOV BX, 0100H
Again:  ADD AL, [BX]
        INC BX
        DEC CX
        JNZ Again
        MOV [0200], AL
        HLT

```



**Example:** Write a program to move a block of 100 consecutive bytes of data starting at offset address 400H in memory to another block of memory locations starting at offset address 600H. Assume both block at the same data segment F000H.

```

MOV AX, F000H
MOV DS, AX
MOV SI, 0400H
MOV DI, 0600H
MOV CX, 64H  ⇝ 64 Hexadecimal = 100 Decimal
LableX: MOV AH, [SI]           ⌂
        MOV [DI], AH
        INC SI
        INC DI
        DEC CX
        JNZ LableX
        HLT   ← End of program
    
```

**Example:** Write a program that counts the number of 1's in a byte and writes it into BL.

```
DATA1 DB 97 ; 61h
SUB BL,BL ;clear BL to keep the number of 1s
MOV DL,8 ;rotate total of 8 times
MOV AL,DATA1
AGAIN: ROL AL,1 ;rotate it once
JNC NEXT ;check for 1
INC BL ; if CF=1 then add one to count
NEXT: DEC DL ;go through this 8 times
JNZ AGAIN ;if not finished go back
HLT
```

Q + T / C A T ; 61h  
SUB BL,BL ;clear BL to keep the number of 1s

**Example** Assume that the port address 22H is an input port for monitoring the temperature. Write Assembly language instructions to monitor the port continuously for the temperature of 100 degrees. If it reaches 100, then BH should contain 'Y'.

```
BACK: IN AL,22H ;get the temperature data from port# 22H
    CMP AL,100 ;is temp =100?
    JNZ BACK ;if not, keep monitoring
    MOV BH,'Y' ;temp =100, load 'Y' into BH
```

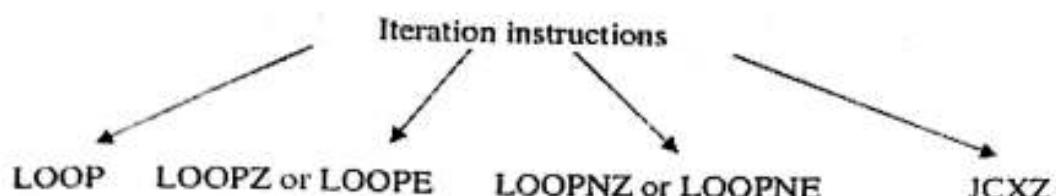
### **Example:**

Write a program to find the highest among 5 grades and write it in DH.

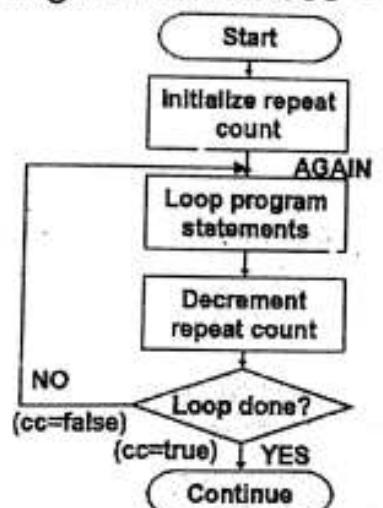
```
DATA DB 51,44,99,88,80
MOV CX,5 ;set up loop counter
MOV BX,OFFSET DATA ;BX points to GRADE data
SUB AL,AL ;AL holds highest grade found so far
AGAIN: CMP AL,[BX] ;compare next grade to highest
    JA NEXT ;jump if AL still highest
    MOV AL,[BX] ;else AL holds new highest
NEXT: INC BX ;point to next grade
    DEC CX
    JNZ AGAIN
    MOV DH, AL
    HLT
```

### Iteration Instructions

Iteration instructions provide a convenient way to implement loops in a program.



## Loop program structures – REPEAT-UNTIL



#### REPEAT-UNTIL program sequence

## LOOP Instruction

- LOOP = DEC CX and JNZ disp
- if CX != 0, jump to the address indicated by the label
- otherwise CX==0, execute the next sequential instruction.



Let us say, we want to repeat a set of instructions 5 times.

For 8085 processor

```
    MVI C, 05H  
AGAIN: MOV B, D  
        :  
        DCR C  
        JNZ AGAIN
```

For 8086 processor

```
    MOV CX, 0005H  
AGAIN: MOV BX, DX  
        :  
        LOOP AGAIN
```



General format: LOOP r8; r8 is 8-bit signed value. It is a 2 byte instruction.

Used for backward jump only. Maximum distance for backward jump is only 128 bytes.  
LOOP AGAIN is almost same as:

```
    DEC CX  
    JNZ AGAIN
```

*Loop Instruction*

LOOP instruction does not affect any flags.

If CX value before entering the iterative loop is:

0005, then the loop is executed 5 times till CX becomes 0

0001, then the loop is executed 1 time till CX becomes 0

## LOOPZ instruction

LOOP while Zero is a 2-byte instruction. It is used for backward jump only. Backward jump takes place if after decrement of CX it is still not zero AND Z flag = 1. LOOPE is same as LOOPZ. LOOPE is abbreviation for LOOP while Equal. LOOPE is normally used after a compare instruction.

Ex.           MOV CX, 04H

BACK:   SUB BX, AX  
          MOV BX, DX

:

:

ADD SI, DI

LOOPZ BACK ; if SI+DI = 0 and CX not equal to 0, branch to BACK

### JCXZ Instruction

Jump if CX is Zero is useful for terminating the loop immediately if CX value is 0000H. It is a 2 byte instruction. It is used for forward jump only. Maximum distance for forward jump is only 127 bytes.

Ex.

AGAIN:	MOV CX, SI JCXZ SKIP MOV BX, DX : : LOOP AGAIN
SKIP:	ADD SI, DI ; Executed after JCXZ if CX = 0

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label	$(CX) \leftarrow (CX)-1$ Jump is initiated to location defined by short-label if $(CX) \neq 0$ ; otherwise, execute next sequential instruction
LOOPE LOOPZ	Loop while equal/loop while zero	LOOPE/LOOPZ short-label	$(CX) \leftarrow (CX)-1$ Jump to location defined by short-label if $(CX) \neq 0$ and $ZF=1$ ; otherwise, execute next sequential instruction
LOOPNE LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ short-label	$(CX) \leftarrow (CX)-1$ Jump to location defined by short-label if $(CX) \neq 0$ and $ZF=0$ ; otherwise, execute next sequential instruction

**Example:** Write a program to move a block of 100 consecutive bytes of data starting at offset address 400H in memory to another block of memory locations starting at offset address 600H. Assume both block at the same data segment F000H.

```

MOV AX, F000H
MOV DS, AX
MOV SI, 0400H
MOV DI, 0600H
MOV CX, 64H
NEXTPT: MOV AH, [SI]
          MOV [DI], AH
          INC SI
          INC DI
          LOOP NEXTPT
          HLT

```

**Example** Write an ALP to find the maximum value of a byte from a string of bytes.

MOV SI, 3000 H	: Source address put in SI
MOV CX, 0100 H	: Count value of bytes put in CX
MOV AH, 00 H	: AH initialised with 00H
XX: CMP AH, [SI]	: AH compared with data pointed to by SI
JAE 200E H	: Jump if AH is $\geq$ (SI) to 200E H
MOV AH, [SI]	: Otherwise, move (SI) to AH
INC SI	: Increment SI
LOOPNZ XX	: Loop unless CX $\neq$ 0
MOV [SI], AH	: (AH) transferred to the memory location pointed to by SI
HLT : Stop.	

**Example** Write an ALP to find the minimum value of a byte from a string of bytes.

MOV SI, 3000 H	: Source address put in SI
MOV CX, 0100 H	: Count value of bytes put in CX
MOV AH, 00 H	: AH initialised with 00H
XX: CMP AH, [SI]	: AH compared with data pointed to by SI
JB 200E H	: Jump if (AH) < (SI) to 200E H
MOV AH, [SI]	: Otherwise move (SI) to AH
INC SI	: Increment SI
LOOPNZ XX	: Loop unless CX $\neq$ 0
MOV [SI], AH	: (AH) transferred to the memory location pointed to by SI
HLT	: Stop.

## PROCEDURES

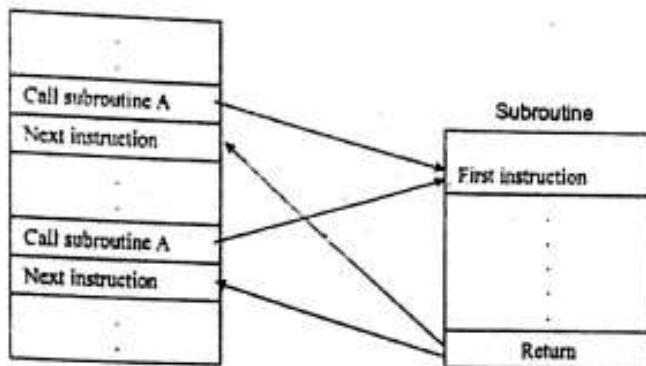
- A procedure is a group of instructions that usually performs one task.
- subroutine, method, or function is an important part of any system's architecture.
- A procedure is a reusable section of the software stored in memory once, but used as often as necessary.
- Disadvantage of procedure is time it takes the computer to link to, and return from it.
- CALL links to the procedure; the RET (return) instruction returns from the procedure.
- CALL pushes the address of the instruction following the CALL (return address) on the stack.
- the stack stores the return address when a procedure is called during a program
- RET instruction removes an address from the stack so the program returns to the instruction following the CALL.
- A procedure begins with the PROC directive and ends with the ENDP directive. each directive appears with the procedure name PROC is followed by the type of procedure: NEAR or FAR.
- Procedures that are to be used by all software (global) should be written as far procedures. Procedures that are used by a given task (local) are normally defined as near procedures

SUMS      PROC NEAR

ADD AX, BX  
ADD AX, CX  
ADD AX, DX  
RET

SUMS      ENDP

## CALL Instructions



Format: **CALL OPERAND**

OPERAND
Near proc
Far proc
Mem ptr
Reg16

e.g.

CALL 1040h ; PUSH IP, (IP)  $\leftarrow$  (IP)+1040h

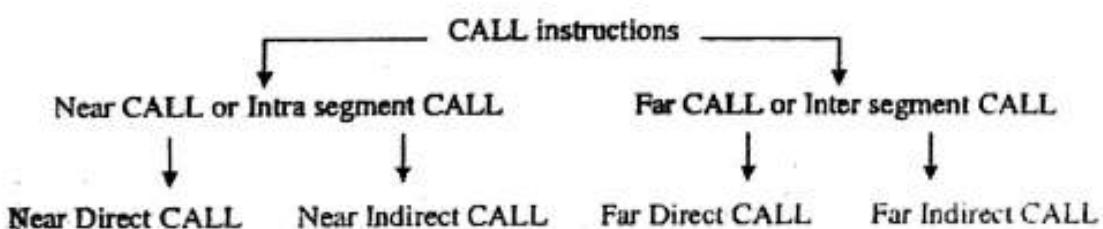
CALL 4060:0280 ; PUSH CS, PUSH IP,  
(IP)  $\leftarrow$  0280h, (CS)  $\leftarrow$  4060h

CALL BX ; PUSH IP, (IP)  $\leftarrow$  (BX)

CALL NEAR PTR[BX] ; PUSH IP, (IP)  $\leftarrow$  [DS: (BX)]

CALL FAR PTR[SI+08h] ; PUSH CS, PUSH IP,  
(IP)  $\leftarrow$  [DS: (SI)+08], (CS)  $\leftarrow$  [DS: (SI)+10]

- CALL instruction is used to branch to a subroutine. There are no conditional Call instructions in 8086.



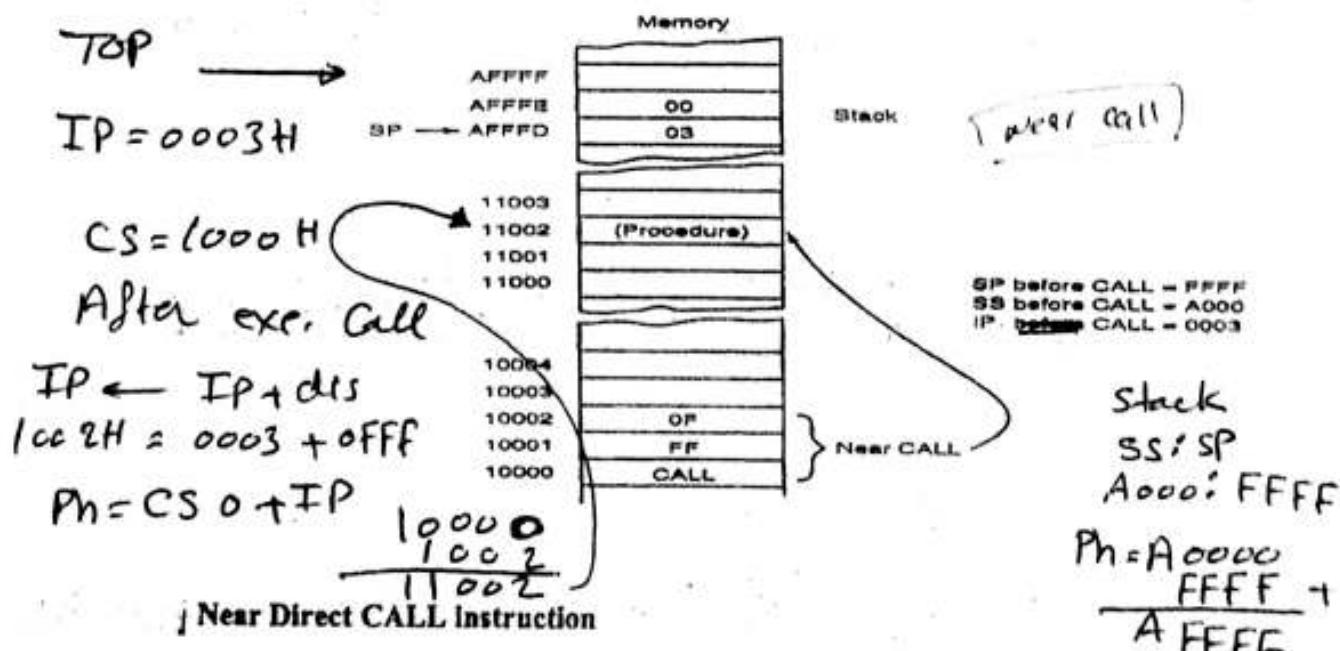
## Call

- Transfers the flow of the program to the procedure.
- CALL instruction differs from the jump instruction because a CALL **saves a return address on the stack**.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.

**Near Call**

- 3 bytes long. The first byte contains the opcode, the second and third bytes contain the displacement.
- When the near CALL executes, it first pushes the offset address of the next instruction onto the stack. offset address of the next instruction appears in the instruction pointer (IP)
- It then adds displacement from bytes 2 & 3 to the IP to transfer control to the procedure.

$$CS = 1000 \text{ H}$$



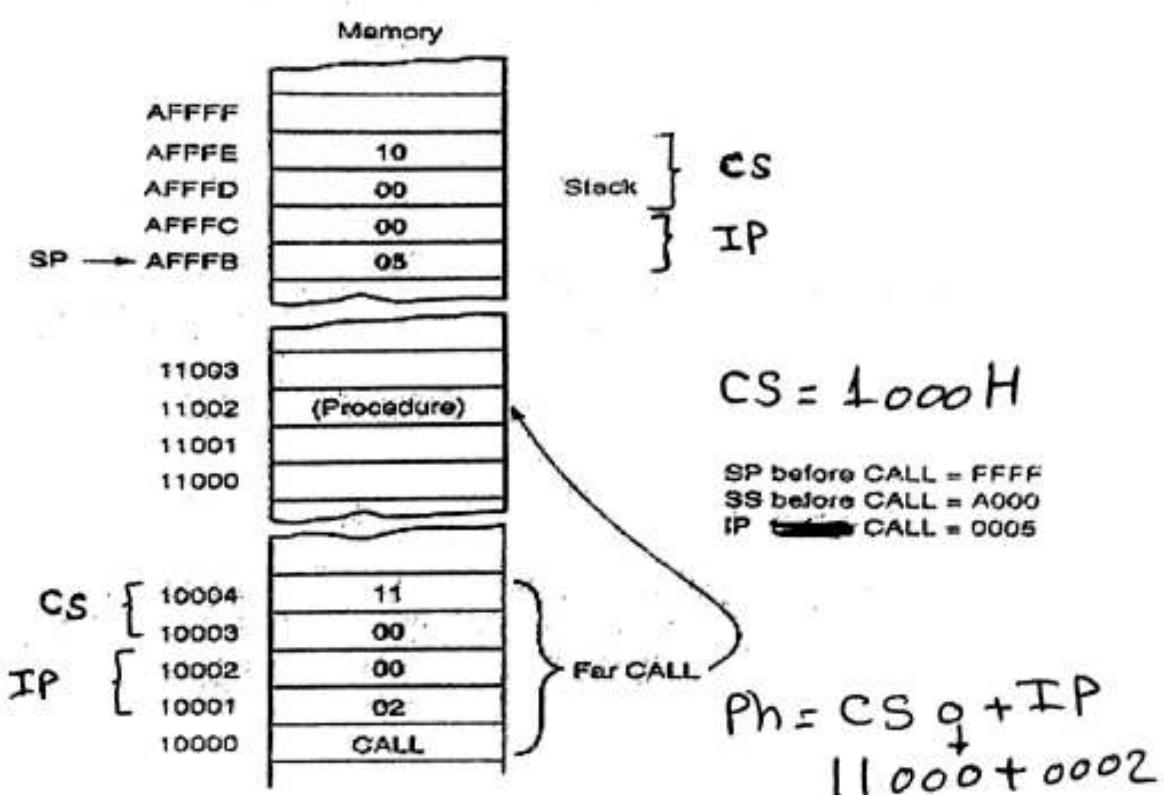
It is a 3-byte instruction. It has the format **CALL r16** and has the range  $\pm 32K$  bytes. Covers the entire Code segment. It is the most common **CALL** instruction.

It is functionally same as the combination of the instructions PUSH IP and ADD IP r16.

### Ex. CALL, Compute

### Far Call

- 5-byte instruction contains an opcode followed by the next value for the IP and CS registers.
- bytes 2 and 3 contain new contents of the IP
- bytes 4 and 5 contain the new contents for CS
- Far CALL places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2 through 5.
- This allows far CALL to call a procedure located anywhere in the memory and return from that procedure.



### Far Direct CALL instruction

- It is a 5-byte instruction. 1-byte opcode, 2-byte offset, 2-byte segment value. Far direct CALL is functionally same as:

PUSH CS  
PUSH IP

$$\begin{array}{r}
 11000 \\
 + 0002 \\
 \hline
 11002
 \end{array}$$

IP = 2-byte offset value provided in CALL  
CS = 2-byte segment value provided in CALL

Ex. CALL far ptr Compute

## CALLs with Register Operands

- *Near Indirect instruction*

An example CALL BX, which pushes the contents of IP onto the stack, then jumps to the offset address, located in register BX, in the current code segment. Always uses a 16-bit offset address, stored in any 16-bit register except segment registers.

Ex. 1: CALL BX

; If (BX) = 1234H, branches to procedure at CS: 1234H.  
1234H is not relative displacement.

Ex. 2: CALL word ptr 2000H[BX]

If BX contents is 1234H Branches to subroutine at CS:5678H

DS:3234H	5678H
DS:3236H	ABCDH

- *Far Indirect instruction*

Ex. CALL dword ptr 2000H[BX]

If BX contents is 1234H Branches to subroutine at ABCDH:5678H

DS:3234H	5678H	+ <u>2000</u> H
DS:3236H	ABCDH	

3234H

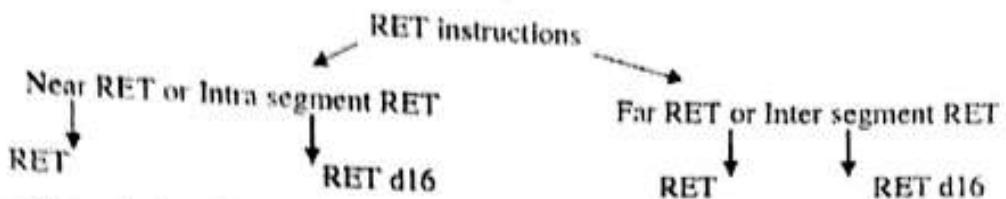
Conditional CALL?

What if we want to branch to subroutine COMPUTE only if Cy flag = 0?

*Solution:*

```
JC NEXT
CALL COMPUTE ; execute only if Cy = 0
NEXT:
```

### RETURN Instructions



**RET** is abbreviation for *Return from subroutine*

Removes a 16-bit number (near return) from the stack placing it in IP, or removes a 32-bit number (far return) and places it in JP & CS.

Format: RET [OPERAND]

<b>OPERAND</b>
None
Disp16

e.g.

RET (near):

RET (far):

RET (far);

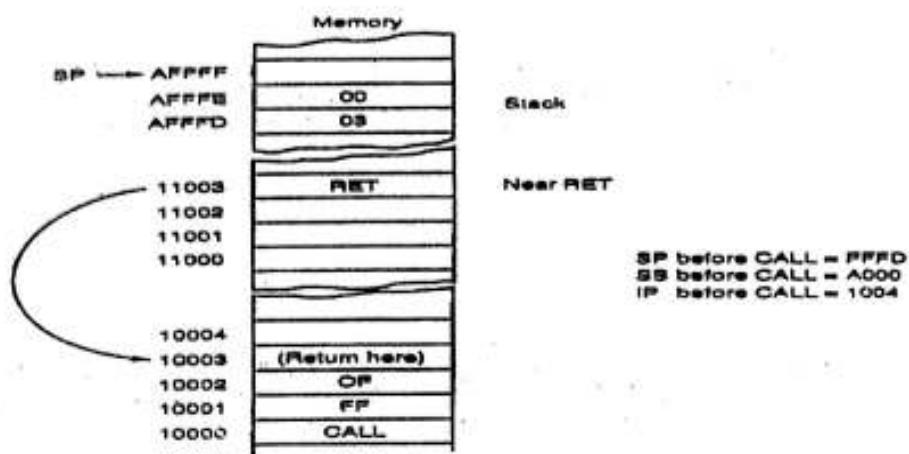
RET 10h (far):

## POP IP

### POP IP, POP CS

POP IP, (SP)  $\leftarrow$  (SP)+10h

POP IP, POP CS, (SP)  $\leftarrow$  (SP)+10h



### Near RET instruction

It is 1-byte instruction. Opcode is C3H. It is functionally same as : POP IP

Fy:

Compute Proc Near : indicates it is a NEAR procedure

10

1

RET

Compute ENDP ; end of procedure Compute

### Far RET instruction

It is 1-byte instruction. Opcode is CBH. It is functionally same as: POP IP + POP CS

Ex.

```
SINX  Proc Far ; indicates it is a FAR procedure
      :
      :
      RET
SINX  ENDP ; end of procedure SINX
```

**Example:** write a procedure named *Square* that squares the contents of BL and places the result in BX.

**Solution:**

*Square:* PUSH AX  
MOV AL, BL  
MUL BL  
MOV BX, AX  
POP AX  
RET

**Example:** write a program that computes  $y = (AL)^2 + (AH)^2 + (DL)^2$ , places the result in CX. Make use of the **SQUARE** subroutine defined in the previous example. (Assume result y doesn't exceed 16 bit)

**Solution:**

```
MOV CX, 0000H
MOV BL, AL
CALL Square
ADD CX, BX
MOV BL, AH
CALL Square
ADD CX, BX
MOV BL, DL
CALL Square
ADD CX, BX
HLT
```

## STRINGS AND STRING-HANDLING INSTRUCTIONS

- Instructions for moving large blocks of data or *strings*.
- **String**: A series of data words (or bytes) that reside in consecutive memory locations
- Each allows data transfers as a single byte, word, or doubleword.
- Before the string instructions are presented, the operation of the D flag-bit (direction), DI, and SI must be understood as they apply to the string instructions.

### The Direction Flag

- The direction flag (D, located in the flag register) selects the autoincrement or the auto-decrement operation for the DI and SI registers during string operations.
  - used only with the string instructions
- STD set direction flag so that the pointers are auto decremented.
- CLD clear direction flag so that the pointers are auto incremented.

### DI and SI

- During execution of string instruction, memory accesses occur through DI and SI registers.
  - DI offset address accesses data in the extra segment for all string instructions that use it
  - SI offset address accesses data by default in the data segment

**String**: A series of data words (or bytes) that reside in consecutive memory locations Permits operations:

- Move data from one block of memory to a block elsewhere in memory,
- Scan a string of data elements stored in memory to look for a specific value,
- Compare two strings to determine if they are the same or different.

**Five basic String Instructions** define operations on this element of a string:

Move byte or word string	MOVSB/MOVSW
Compare string	CMPSB/CMPSW
Scan string	SCANB/SCANW
Load string	LODHB/LODSW
Store string	STOSB/STOSW

Repetition is needed to handle more than one element of a string.

Mnemonic	Meaning	Format	Operation	Flags affected
MOVSB 	Move string	MOVSB MOVSW	$((ES)0+(DI)) \leftarrow ((DS)0+(SI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	None
CMPSB	Compare string	CMPSB CMPSW	$((DS)0+(SI)) - ((ES)0+(DI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	CF, PF, AF, ZF, SF, OF
SCASB	Scan string	SCASB SCASW	$(AL \text{ or } AX) - ((ES)0+(DI))$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	CF, PF, AF, ZF, SF, OF
LODSB	Load string	LODSB LODSW	$(AL \text{ or } AX) \leftarrow ((DS)0+(SI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$	None
STOSB	Store string	STOSB STOSW	$((ES)0+(DI)) \leftarrow (AL \text{ or } AX)$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	None

## MOV\$

Transfers data from a memory location to another

This is the only memory-to- memory transfer allowed.

The MOVS instruction transfer a byte or a word from the data segment location addressed by the SI to the extra segment location addressed by the DI. The pointers then increment or decrement according to DF.

MOVSB	ES:[DI]	←	DS:[SI]
	If DF = 0, SI	←	SI+1
	DI	←	DI+1
	If DF = 1, SI	←	SI-1
	DI	←	DI-1
MOVSW	ES:[DI]	←	DS:[SI]
	ES:[DI+1]	←	DS:[SI+1]
	If DF = 0, SI	←	SI+2
	DI	←	DI+2
	If DF = 1, SI	←	SI-2
	DI	←	DI-2

## LODS

Loads AL or AX with the data stored at the data segment memory location addressed by the SI register; if DF = 0, increment SI, else decrement SI

LODSB      AL       $\leftarrow$       DS:[SI]

If DF = 0, SI       $\leftarrow$       SI+1

If DF = 1, SI       $\leftarrow$       SI-1

LODSW      AL       $\leftarrow$       DS:[SI]

AH       $\leftarrow$       DS:[SI+1]

If DF = 0, SI       $\leftarrow$       SI+2

If DF = 1, SI       $\leftarrow$       SI-2

## STOS

Stores AL or AX at the extra segment memory location addressed by the DI register; if DF = 0, increment DI, else decrement DI

STOSB      ES:[DI]       $\leftarrow$       AL

If DF = 0, DI       $\leftarrow$       DI+1

If DF = 1, DI       $\leftarrow$       DI-1

STOSW      ES:[DI]       $\leftarrow$       AL

ES:[DI+1]       $\leftarrow$       AH

If DF = 0, DI       $\leftarrow$       DI+2

If DF = 1, DI       $\leftarrow$       DI-2

## String Instructions

- String is a collection of bytes, words, or long-words that can be up to 65535 in length
- String instructions can have at most two operands. One is referred to as source string and the other one is called destination string
  - Source string must locate in Data Segment and SI register points to the current element of the source string
  - Destination string must locate in Extra Segment and DI register points to the current element of the destination string

Source String	Destination String
0511:0001	02AS:2000
0511:0001	02AS:2001
0511:0002	02AS:2002
0511:0003	02AS:2003
0510:0004	02AS:2004
0510:0005	02AS:2005
0511:0006	02AS:2006

### Repeat Prefix Instructions

### 1 REP Series Instruction

- The prefix instruction makes the microprocessor repeatedly execute the string instruction until CX decrements to 0 (During the execution, CX is decreased by one when the string instruction is executed one time).
- For Example:

MOV CX, 5  
REP MOVSB

By the above two instructions, the microprocessor will execute MOVSB 5 times.

#### Execution flow of REP MOVSB::

```
While (CX!=0)
  CX = CX-1;      OR
  MOVSB;
  /
```

```

Check_CX: If CX!=0 Then
            CX = CX-1;
            MOVSBL;
            goto Check_CX;
        end if

```

## Repeat Prefix Instructions

- REPZ *String Instruction*
  - Repeat the execution of the string instruction until CX=0 or zero flag is clear
- REPNZ *String Instruction*
  - Repeat the execution of the string instruction until CX=0 or zero flag is set
- REPE *String Instruction*
  - Repeat the execution of the string instruction until CX=0 or zero flag is clear
- REPNE *String Instruction*
  - Repeat the execution of the string instruction until CX=0 or zero flag is set

## Direction Flag

- Direction Flag (DF) is used to control the way SI and DI are adjusted during the execution of a string instruction
  - DF=0, SI and DI will auto-increment during the execution; otherwise, SI and DI auto-decrement
  - Instruction to set DF: **STD**; Instruction to clear DF: **CLD**

— Example:

CLD  
MOV CX, 5  
REP MOVS B

At the beginning of execution,  
DS=0510H and SI=0000H

DS : SI		
0510:0000	53	S ← SI $CX=5$
0510:0001	48	H ← SI $CX=4$
0510:0002	4F	O ← SI $CX=3$
0510:0003	50	P ← SI $CX=2$
0510:0004	50	P ← SI $CX=1$
0510:0005	45	E ← SI $CX=0$
0510:0006	52	R

Source String

### U MOVEMENTS

Move (byte read) of memory location 100-01 to memory location 100-00 and update 10 and 11 according to 100 and the state of the data being transferred. It does not modify flags.

Source String	Destination String
MOV AX,0310H	0310-0000
MOV BX, AX	0310-0001
MOV CL, 0	0310-0002
MOV AX, 0300H	0310-0003
MOV ES, AX	0310-0004
MOV DL, 100H	0310-0005
CLD	0310-0006
MOV CX, 5	0310-0007
REP MOVSB	0310-0008

#### CMPSB (MPSW)

Compare bytes (words) at memory locations DS:SI and ES:DI.  
update SI and DI according to DF and the width of the data being compared  
It modifies flags

Assume	ES = 02ABH	DS : SI		DS : DI		
	DI = 2000H	0510:0000	33	S	02AB:2000	33
	DS = 0510H	0510:0001	48	H	02AB:2001	48
	SI = 0000H	0510:0002	4F	O	02AB:2002	4F
CLD		0510:0003	30	P	02AB:2003	50
MOV CX, 9		0510:0004	50	P	02AB:2004	50
REPZ CMPSB		0510:0005	45	H	02AB:2005	49
		0510:0006	52	R	02AB:2006	4B

**Example:** Using string operation, write a program to move a block of 100 consecutive bytes of data starting at offset address 4001H in memory to another block of memory locations starting at offset address 6001H. Assume both block at the same data segment F000H.

```
MOV CX,64H  
MOV AX,1000H
```

```

MOV DS, AX
MOV ES, AX
MOV SI, 1000H
MOV DI, 6000H
CLD
NEXT: STOSB
LOOP NEXT
HLT

```

**Note :** The direction flag selects the auto increment ( $D=0$ ) or the autodecrement ( $D=1$ ) operation for the DI and SI registers during string operations.

**Example** Write a program loads the block of memory locations from A000H through 6A00FH with number 0F.

**Solution:**

```

MOV AX, 0F
MOV DS, AX
MOV ES, AX
MOV AL, 0F
MOV DI, A000H
MOV CX, 0FFH
CLD
AGAIN: STOSB
J LOOP AGAIN

```

### EXAMPLE

Write the assembly language programming to find the 2's complement for a string of 100 bytes - 8086

The following would be the code:

CLD	: Clears the direction flag
MOV SI, 4000H	: In the SI the store address is placed
MOV DI, 5000H	: In the DI the destination address is put
MOV CX, 0064H	: In the CX the number of bytes to be 2's complemented are placed
XX: LODSB	: The data byte is give to AL and INC SI
NEG AL	: AL of 2's complement.
STOSB	: Current AL value into DI and INC DI
LOOPNZ XX	: The loop is maintained till CX becomes = 0.
HLT	: Stop.

### **Flag Manipulation instructions**

The Flag manipulation instructions directly modify some of the Flags of 8086.

- i. CLC - Clear Carry Flag.
- ii. CMC - Complement Carry Flag.
- iii. STC - Set Carry Flag.
- iv. CLD - Clear Direction Flag.
- v. STD - Set Direction Flag.
- vi. CLI - Clear Interrupt Flag.
- vii. STI - Set Interrupt Flag.

**Example:** Clear the carry flag without using CLC instruction.

**Solution**

STC  
CMC

### **Machine Control instructions**

The Machine control instructions control the bus usage and execution

- i. WAIT - Wait for Test input pin to go low.
- ii. HLT - Halt the process.
- iii. NOP - No operation.
- iv. ESC - Escape to external device like NDP
- v. LOCK - Bus lock instruction prefix.

**HALT Instruction** - The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The only way to get the processor out of the halt state are with an interrupt signal on the INTR pin or an interrupt signal on NMI pin or a reset signal on the RESET Input.

**NOP Instruction** - This instruction simply uses up the three clock cycles and increments the instruction pointer to point to the next instruction. NOP does not change the status of any flag. The NOP instruction is used to increase the delay of a delay loop.

# Assembler Directives

## Assume

**DB** - Defined Byte.  
**DD** - Defined Double Word  
**DQ** - Defined Quad Word  
**DT** - Define Ten Bytes  
**DW** - Define Word

➤ **ASSUME Directive** - The ASSUME directive is used to tell the assembler that the name of the logical segment should be used for a specified segment. The 8086 works directly with only 4 physical segments: a Code segment, a data segment, a stack segment, and an extra segment.

## ➤ Example:

**ASUME CS:CODE** ;This tells the assembler that the logical segment named CODE contains the instruction statements for the program and should be treated as a code segment.

**ASUME DS:DATA** ;This tells the assembler that for any instruction which refers to a data in the data segment, data will be found in the logical segment DATA.

➤ **DB**

DB directive is used to declare a byte-type variable or to store a byte in memory location.

➤ **Example:**

1. **PRICE** DB 49h, 98h, 29h ; Declare an array of 3 bytes, named as PRICE and initialize.
2. **NAME** DB 'ABCDEF' ; Declare an array of 6 bytes and initialize with ASCII code for letters
3. **TEMP** DB 100 DUP(?) ; Set 100 bytes of storage in memory and give it the name as TEMP, but leave the 100 bytes uninitialized. Program instructions will load values into

➤ **DW**

The DW directive is used to define a variable of type word or to reserve storage location of type word in memory.

➤ **Example:**

**MULTIPLIER** DW 437Ah ; this declares a variable of type word and named it as MULTIPLIER. This variable is initialized with the value 437Ah when it is loaded into memory to run.

**EXPI** DW 1234h, 3456h, 5678h ; this declares an array of 3 words and initialized with specified values.

**STOR1** DW 100 DUP(0); Reserve an array of 100 words of memory and initialize all words with 0000. Array is named as STOR1.

DUP directive: once you need to declare a large array you can use DUP.

format:

number DUP (value(s))

number : number of duplicate to make (any constant value).

value : expression that DUP will duplicate.

Example: term DB 7 DUP(5) ; term DB 5,5,5,5,5,5,5

- **ENDS** This ENDS directive is used with name of the segment to indicate the end of that logic segment.
- **Example:**

**CODE**      **SEGMENT** ;Hear it Start the logic segment containing code  
 ; Some instructions statements to perform the logical operation

**CODE**      **ENDS** ;End of segment named as ;CODE

- **END** END directive is placed after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statement after an END directive. Carriage return is required after the END directive.

- **ENDP** ENDP directive is used along with the name of the procedure to indicate the end of a procedure to the assembler

- **Example:**

**SQUARE NUM PROC** ; It start the procedure  
 ; Some steps to find the square root of a number

**SQUARE\_NUM ENDP** ;Hear it is the End for the procedure

- **END** End Program
- **ENDP** End Procedure
- **ENDS** End Segment
- **EQU** Equate
- **EVEN** Align on Even Memory Address

- **EQU** This EQU directive is used to give a name to some value or to a symbol. Each time the assembler finds the name in the program, it will replace the name with the value or symbol you given to that name.

➤ Example:

**FACTOR EQU 03H** ; you has to write this statement at the starting of your program and later in the program you can use this as follows

**ADD AL, FACTOR** ; When it codes this instruction the assembler will code it as ADDAL, 03H : The advantage of using EQU in this manner is, if FACTOR is used many no of times in a program and you want to change the value, all you had to do is change the EQU statement at beginning, it will changes the rest of all.

- **PROC** The PROC directive is used to identify the start of a procedure. The term near or far is used to specify the type of the procedure.

➤ Example:

**SMART PROC FAR** ; This identifies that the start of a procedure named as SMART and instructs the assembler that the procedure is far.

**SMART ENDP**

This PROC is used with ENDP to indicate the break of the procedure.

- **PTR** This PTR operator is used to assign a specific type of a variable or to a label.

➤ Example:

**INC [BX]** ; This instruction will not know whether to increment the byte pointed to by BX or a word pointed to by BX.

**INC BYTE PTR [BX]** ;increment the byte pointed to by BX

- ❖ **Model directive:** - The model directive specifies the total amount of memory the program would take.

**MODEL** mem\_mod ; where mem\_mod can be :  
TINY , SMALL , COMPACT , MEDIUM , LARGE or Huge

Memory Model	Size of Code	Size of Data	Segments
TINY	Code + Data <= 64KB	Code & data combined	
SMALL	Less <= 64KB	Less <= 64KB	1 code seg , 1 data seg
MEDIUM	Any size	Less <= 64 KB	Many code segs , 1 data seg
COMPACT	Less <= 64KB	Any size	1 code seg, many data segs
LARGE	Any size	Any size	Many code and data segs
HUGE	Any size	Any size	Many code and data segs

- ❖ **Segment directives:**

- **Data Segments:** - .data followed by declarations of variables or definitions of constants.
- **Code Segment :** - .code followed by a sequence of program statements.

1. Write an ALP to find factorial of number for 8085.

```
MOV AX, 0SH
MOV CX, AX
DEC CX
Back: MUL CX
LOOP back
MOV [D000], AX
HLT
```

2. Sum of series of 10 numbers and store result in memory location total.

data  
List dw 12,34,56,78,98,01,13,78,18,36  
Total dw ?

code

Main proc

```
MOV AX, @data
MOV DS, AX
MOV AX, 0000H
MOV CX, 0AH ; counter
MOV BX, 0000H ; to count carry
MOV SI, offset List
```

Back : ADD AL, [SI]

JC Label

Back1: INC SI

LOOP Back

MOV Total, AX

MOV Total+2, Bx

Label: INC Bx

JMP Back1

Main endp

End Main

3. Write a program to multiply 2 numbers (16-bit data) for 8086.

```
.data
Multiplier      dw  1234H
Multiplicand    dw  3456H
Product        dw  ?
.code
MULT proc
MOV AX, @data
MOV DS, AX
MOV AX, Multiplicand
MUL Multiplier
MOV Product, AX
MOV Product+2, DX
MULT endp
```

4- Write a program to find Largest No. in a block of data. Length of block is 0A. Store the maximum in location result.

```
; Title maximum in given series
.model small
.stack 100h
.data
List db 80, 81, 78, 65, 23, 45, 89, 90, 10, 99
Result db ?

.code
Main proc
    MOV AX, @data
    MOV DS, AX
    MOV SI, offset List
    MOV AL, 00H
    MOV CX, 0AH
Back:  CMP AL, [SI]
        JNC Ahead
        MOV AL, [SI]
Ahead: INC SI
        LOOP Back
        MOV Result, AL
Main endp
End Main
```

5. Find number of times letter 'e' exist in the string 'exercise', Store the count

Title string operation

```
.model small  
.stack 100h  
.data  
String db 'exercise'  
Ans db ?  
Length equ 8  
.code
```

Main proc

```
    MOV AX, @data  
    MOV DS, AX  
    MOV AL, 00H  
    MOV SI, offset String  
    MOV CX, Length  
Back:  MOV BH, [SI]  
        CMP BH, 'e'  
        JNZ Label  
        INC AL  
Label: INC SI  
        LOOP Back  
        MOV Ans, AL  
Main endp  
End Main
```